A* algorithm

Q) 8 puzzle using A* with Manhattan distance & misplaced tiles.

5	4	
6	1	8
7	3	2

Initial state

1	2	3
4	5	6
7	8	

Goal state

→ 1) Manhattan distance

Algorithm

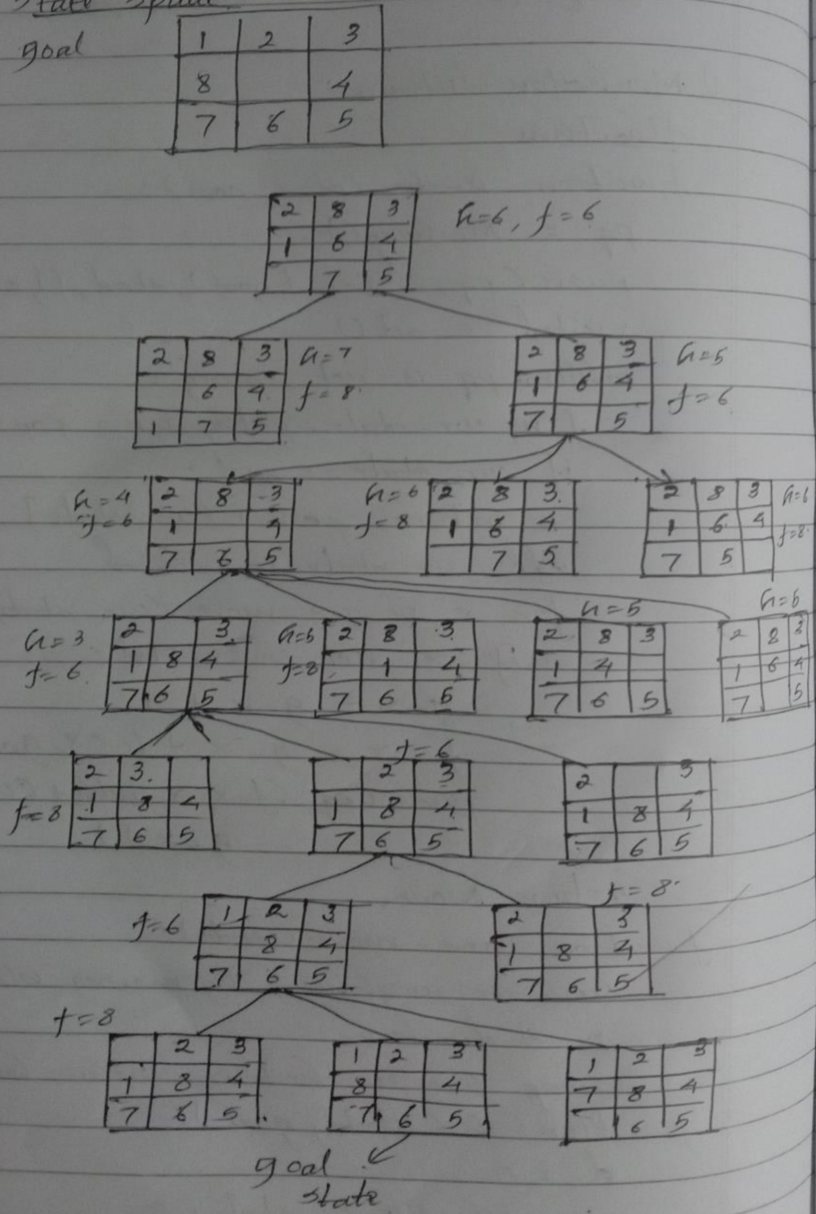
```
function a-star(start, goal):
    pq = MinHeap()
    push(pq, (h(start, goal), start, [], 0))
    visited = set()
    while pq is not empty:
        f-n, cur-state, path, g-n = pop(pq)
        if cur-state == goal:
            return path + [cur-state]
        add cur-state to visited
        for x in gen-moves(cur-state):
            if x not in visited:
                g-x = g-n + 1
                f-x = g-x + h(x, goal)
                push(pq, (f-x, x, path + [cur-state],
                          g-x))
    return None.
function gen-moves(state):
    Generate neighbours by moving blank tile
    in 4 directions
    swap 0 with the tiles within bounds.
function h(state, goal):
    d = 0
    for each tile in state:
```

Manhattan distance as Heuristic

State space tree

```
if tile! = 0:
    distance += abs(current_row - goal_row) +
        abs(current_col - goal_col)
return distance.
```

State space



Code

```
import heapq

# Function to print the puzzle in a 3x3 grid format
def print_puzzle(state):
    for i in range(3):
        print(state[i * 3:(i + 1) * 3])
    print()

# Manhattan Distance Heuristic (h)
def h(state, goal):
    manhattan_distance = 0
    for i in range(9):
        if state[i] != 0:
            current_row, current_col = i // 3, i % 3
            goal_index = goal.index(state[i])
            goal_row, goal_col = goal_index // 3, goal_index % 3
            manhattan_distance += abs(current_row - goal_row) +
abs(current_col - goal_col)
    return manhattan_distance

# Function to check if a given state is the goal state
def is_goal(state, goal):
    return state == goal

# Function to find the index of the blank tile (0) in the puzzle state
def find_blank_tile(state):
    return state.index(0)

# Function to generate all possible moves from a given state
def generate_moves(state):
    neighbors = []
    # Directions are represented as: (row_change, col_change)
    directions = {
        'up': -3,     # Move up by subtracting 3 (index change)
        'down': 3,    # Move down by adding 3 (index change)
        'left': -1,   # Move left by subtracting 1
        'right': 1    # Move right by adding 1
    }

    blank_index = find_blank_tile(state)

    for move, position_change in directions.items():
        new_blank_index = blank_index + position_change

        # Check if the new position is within the bounds
        if move == 'up' and blank_index // 3 == 0:
            continue
```

```
        if move == 'down' and blank_index // 3 == 2:
            continue
        if move == 'left' and blank_index % 3 == 0:
            continue
        if move == 'right' and blank_index % 3 == 2:
            continue

        # Swap the blank tile with the adjacent tile to generate a new state
        new_state = state[:]
        new_state[blank_index], new_state[new_blank_index] =
new_state[new_blank_index], new_state[blank_index]
        neighbors.append(new_state)

    return neighbors

# A* Algorithm
def a_star(start, goal):
    # Priority queue to store (f(n), current_state, path, g(n))
    priority_queue = []
    heapq.heappush(priority_queue, (h(start, goal), start, [], 0))  # f(n),
state, path, g(n)
    visited = set()

    while priority_queue:
        f_n, current_state, path, g_n = heapq.heappop(priority_queue)

        if is_goal(current_state, goal):
            return path + [current_state]  # Return the path to the goal state

        visited.add(tuple(current_state))

        # Generate all possible moves
        for neighbor in generate_moves(current_state):
            if tuple(neighbor) not in visited:
                g_neighbor = g_n + 1  # Increment g(n) for the neighbor
                f_neighbor = g_neighbor + h(neighbor, goal)  # f(n) = g(n) +
h(n)
                heapq.heappush(priority_queue, (f_neighbor, neighbor, path +
[current_state], g_neighbor))

    return None  # No solution found

# Define the start and goal states as flat lists
start_state = [2, 8, 3, 1, 6, 4, 0, 7, 5]
goal_state = [1, 2, 3, 8, 0, 4, 7, 6, 5]

# Perform A* to solve the puzzle
solution_path = a_star(start_state, goal_state)
```

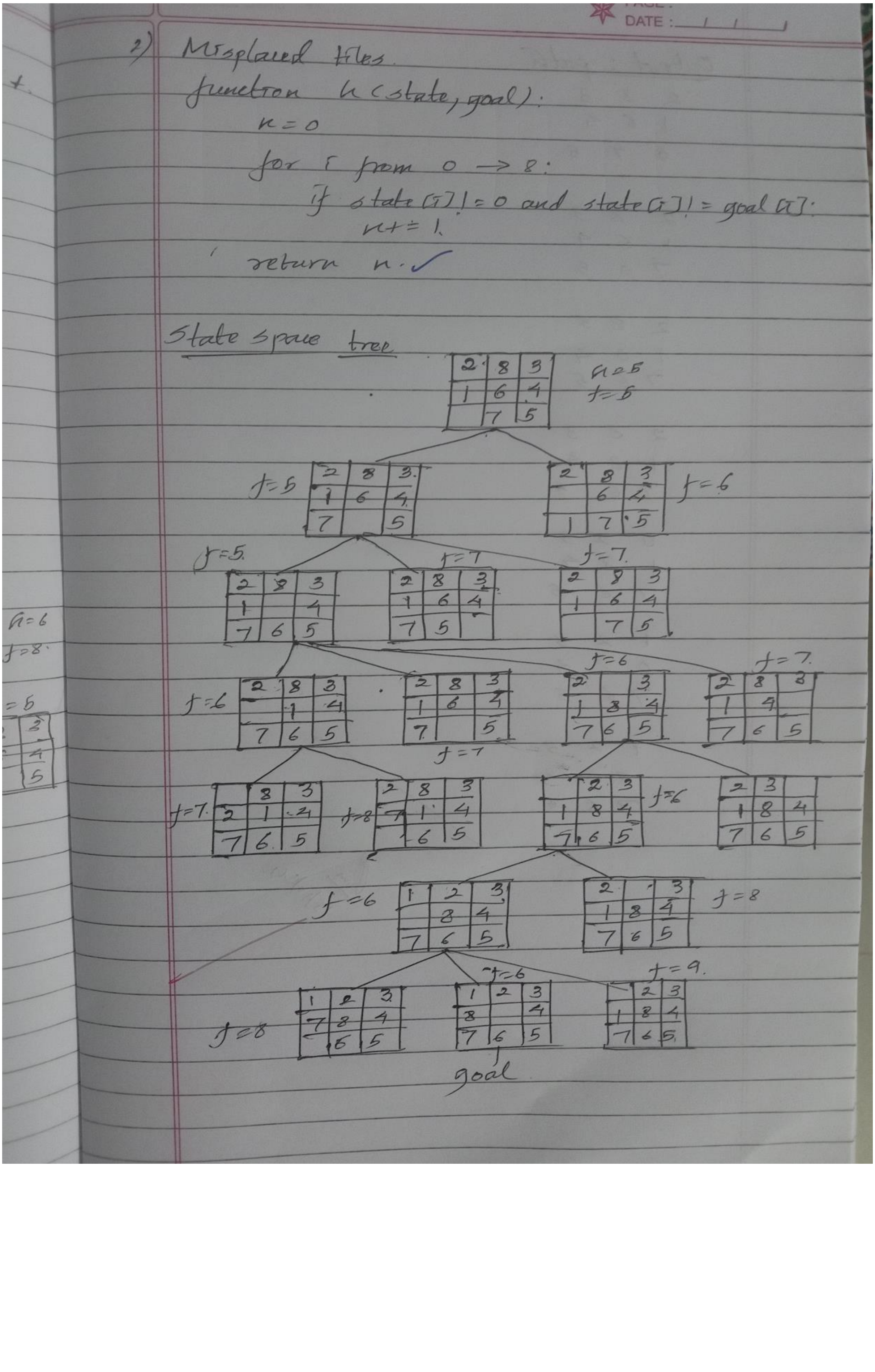
```python
# Display the solution
if solution_path:
    print(f"Solution found in {len(solution_path) - 1} moves:\n")
    for step in solution_path:
        print_puzzle(step)
else:
    print("No solution found.")
```

Output:

```
Solution found in 6 moves:

[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
...
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```

Misplaced Tiles as heuristic

State space tree



2) Misplaced tiles.

```
function h(state, goal):
    n = 0
    for i from 0 -> 8:
        if state[i] != 0 and state[i] != goal[i]:
            n += 1
    return n.
```

State space tree

Code

```
import heapq

# Function to print the puzzle in a 3x3 grid format
def print_puzzle(state):
    for i in range(3):
        print(state[i * 3:(i + 1) * 3])
    print()

# Manhattan Distance Heuristic (h)
def h(state, goal):
    return sum(1 for i in range(9) if state[i] != 0 and state[i] != goal[i])


# Function to check if a given state is the goal state
def is_goal(state, goal):
    return state == goal

# Function to find the index of the blank tile (0) in the puzzle state
def find_blank_tile(state):
    return state.index(0)

# Function to generate all possible moves from a given state
def generate_moves(state):
    neighbors = []
    # Directions are represented as: (row_change, col_change)
    directions = {
        'up': -3,     # Move up by subtracting 3 (index change)
        'down': 3,    # Move down by adding 3 (index change)
        'left': -1,   # Move left by subtracting 1
        'right': 1    # Move right by adding 1
    }

    blank_index = find_blank_tile(state)

    for move, position_change in directions.items():
        new_blank_index = blank_index + position_change

        # Check if the new position is within the bounds
        if move == 'up' and blank_index // 3 == 0:
            continue
        if move == 'down' and blank_index // 3 == 2:
            continue
        if move == 'left' and blank_index % 3 == 0:
            continue
        if move == 'right' and blank_index % 3 == 2:
            continue
```

```
        # Swap the blank tile with the adjacent tile to generate a new state
        new_state = state[:]
        new_state[blank_index], new_state[new_blank_index] =
new_state[new_blank_index], new_state[blank_index]
        neighbors.append(new_state)

    return neighbors

# A* Algorithm
def a_star(start, goal):
    # Priority queue to store (f(n), current_state, path, g(n))
    priority_queue = []
    heapq.heappush(priority_queue, (h(start, goal), start, [], 0))  # f(n),
state, path, g(n)
    visited = set()

    while priority_queue:
        f_n, current_state, path, g_n = heapq.heappop(priority_queue)

        if is_goal(current_state, goal):
            return path + [current_state]  # Return the path to the goal state

        visited.add(tuple(current_state))

        # Generate all possible moves
        for neighbor in generate_moves(current_state):
            if tuple(neighbor) not in visited:
                g_neighbor = g_n + 1  # Increment g(n) for the neighbor
                f_neighbor = g_neighbor + h(neighbor, goal)  # f(n) = g(n) +
h(n)
                heapq.heappush(priority_queue, (f_neighbor, neighbor, path +
[current_state], g_neighbor))

    return None  # No solution found

# Define the start and goal states as flat lists
start_state = [2, 8, 3, 1, 6, 4, 0, 7, 5]
goal_state = [1, 2, 3, 8, 0, 4, 7, 6, 5]

# Perform A* to solve the puzzle
solution_path = a_star(start_state, goal_state)

# Display the solution
if solution_path:
    print(f"Solution found in {len(solution_path) - 1} moves:\n")
    for step in solution_path:
        print_puzzle(step)
else:
```

```python
    print("No solution found.")
```

Output:

```
Solution found in 6 moves:

[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
...
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```